A User-Programmable Vertex Engine

Erik Lindholm

Mark Kilgard

Henry Moreton

NVIDIA Corporation

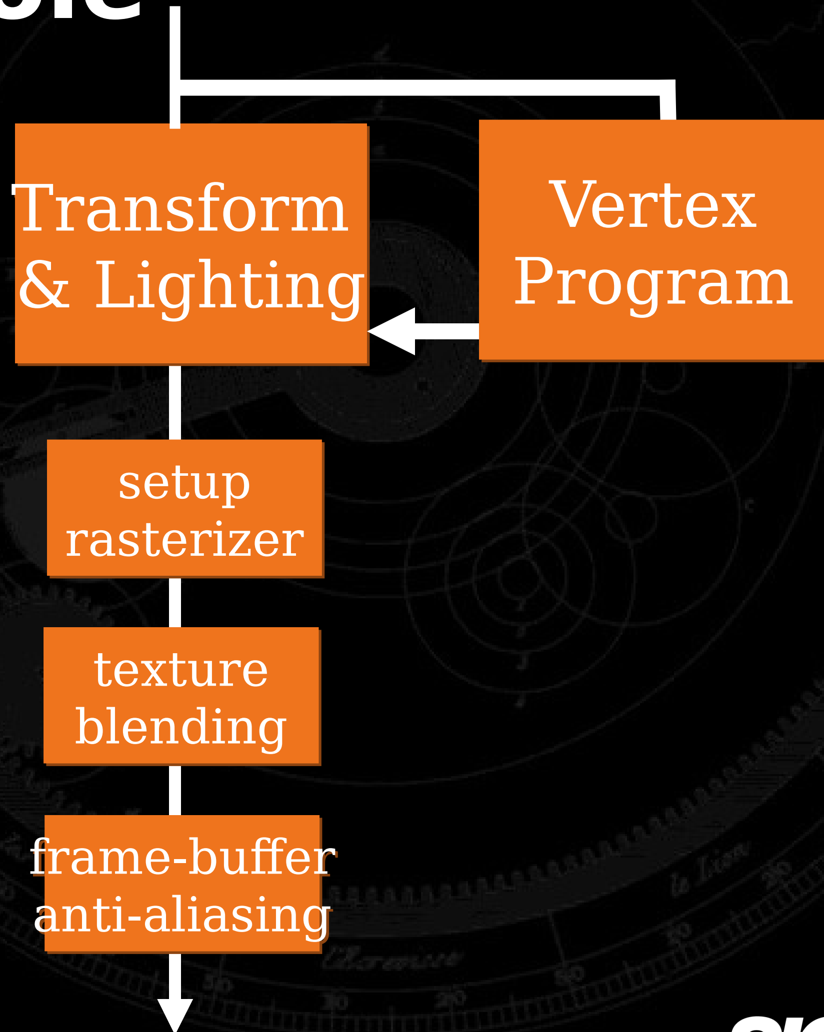
SIGGRAPH 2001

EXPLORE INTERACTION AND DIGITAL IMAGES

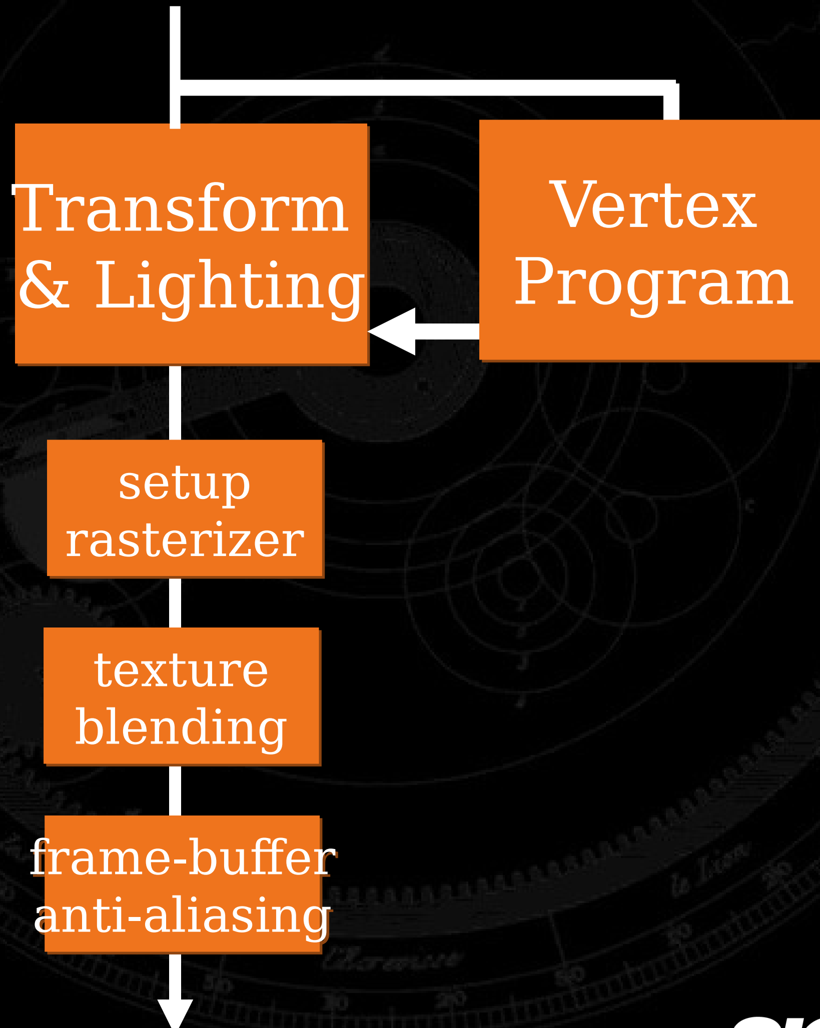
Introduction

- Programmability vs. Configurability
- Programmability is trend
- Very efficient vertex engine
- Simple yet powerful programming model
- Supports fixed function pipeline
- Delivered in the GeForce3

Pipeline Role



Same HW!



nVIDIA GeForce3

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Previous Work: Geometry Engine

- High bandwidth + lots of Flops
- Low clock rate
- No architectural continuity
- VERY hard to program
- Some high-level language support (maybe)
- A compromise solution (vtx,prim,pix, ...)

Alternative: The CPU

- Low bandwidth + reasonable Flops
- High clock rate
- Excellent architectural continuity
- VERY hard to use efficiently
- Excellent high-level language support
- Flexible, but often too slow

NVIDIA GeFORCE3

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

New Design: The Vertex Engine

- Simple hardware for a commodity GPU
- Allows user to manipulate vertex transform
- Simple to use programming model
- Superset of fixed function mode

Why Vertex Processing?

- Very parallel
- Use single vertex programming model
- Hardware can batch or interleave
- KISS

Why Not Primitive Processing?

- Face culling and clipping break parallelism
- Complicates memory accesses
- Inefficient (control takes time)
- Let hardware designers optimize

Programming Model: Vertex I/O

- Streaming vertex architecture
- Source data converted to floats
- Source data loaded
- Run program
- Destination data drained
- Destination data re-formatted for hw

nVIDIA GeForce3

SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Programming Model

Vertex Source
16x4 registers

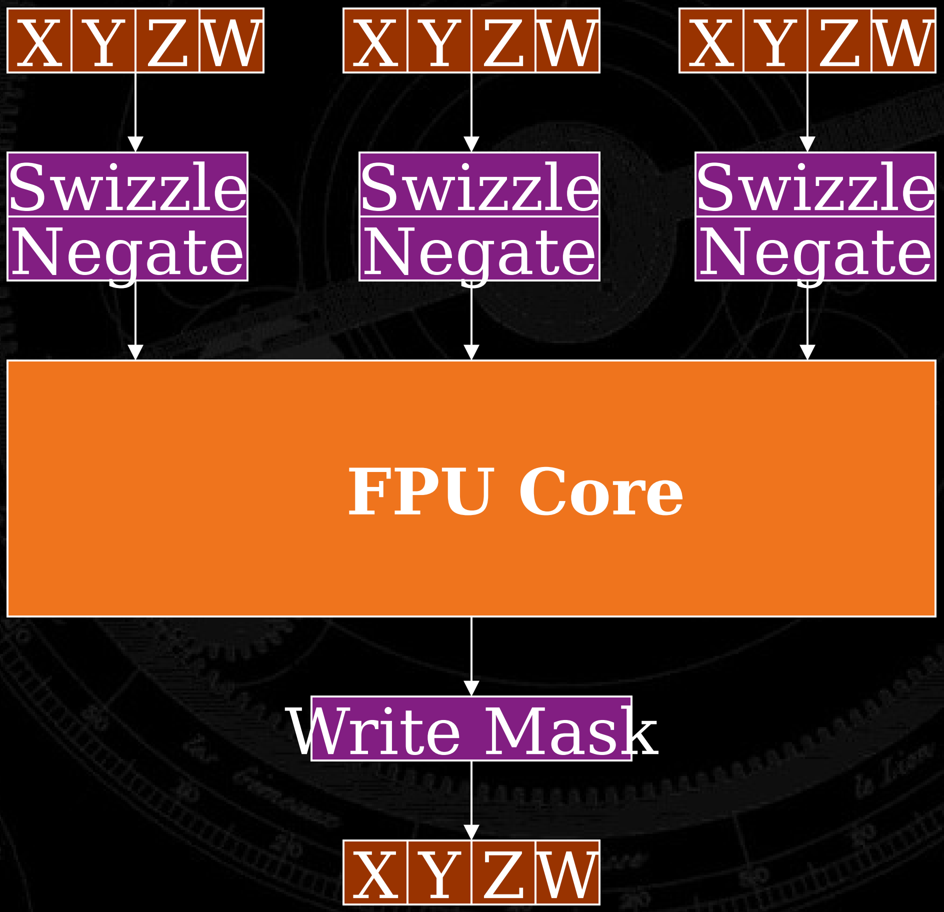
Vertex Program
128 instructions

Vertex Output
15x4 registers

Program Constants
96x4 registers

Temporary Registers
12x4 registers

Data Path



SIGGRAPH 2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Instruction Set: The Core Features

- Immediate access to sources
- Swizzle/negate on all sources
- Write mask on all destinations
- DP3,DP4 most common graphics ops
- Cross product is MUL+MAD with swizzling
- LIT instruction implements phong lighting

Cross Product Coding Example

```
# Cross product R2 = R0 x R1

MUL R2, R0.zxyw, R1.yzxw;
MAD R2, R0.yzxw, R1.zxyw, -R2;
```

Sample OpenGL Vertex Program

```
static const GLubyte vpgm[] = “\!!VP1. 0\
  DP4   o[HPOS].x, c[0], v[0];
    \
  DP4   o[HPOS].y, c[1], v[0];
    \
  DP4   o[HPOS].z, c[2], v[0];
    \
  DP4   o[HPOS].w, c[3], v[0];
    \
  MOV   o[COL0], v[3];
```

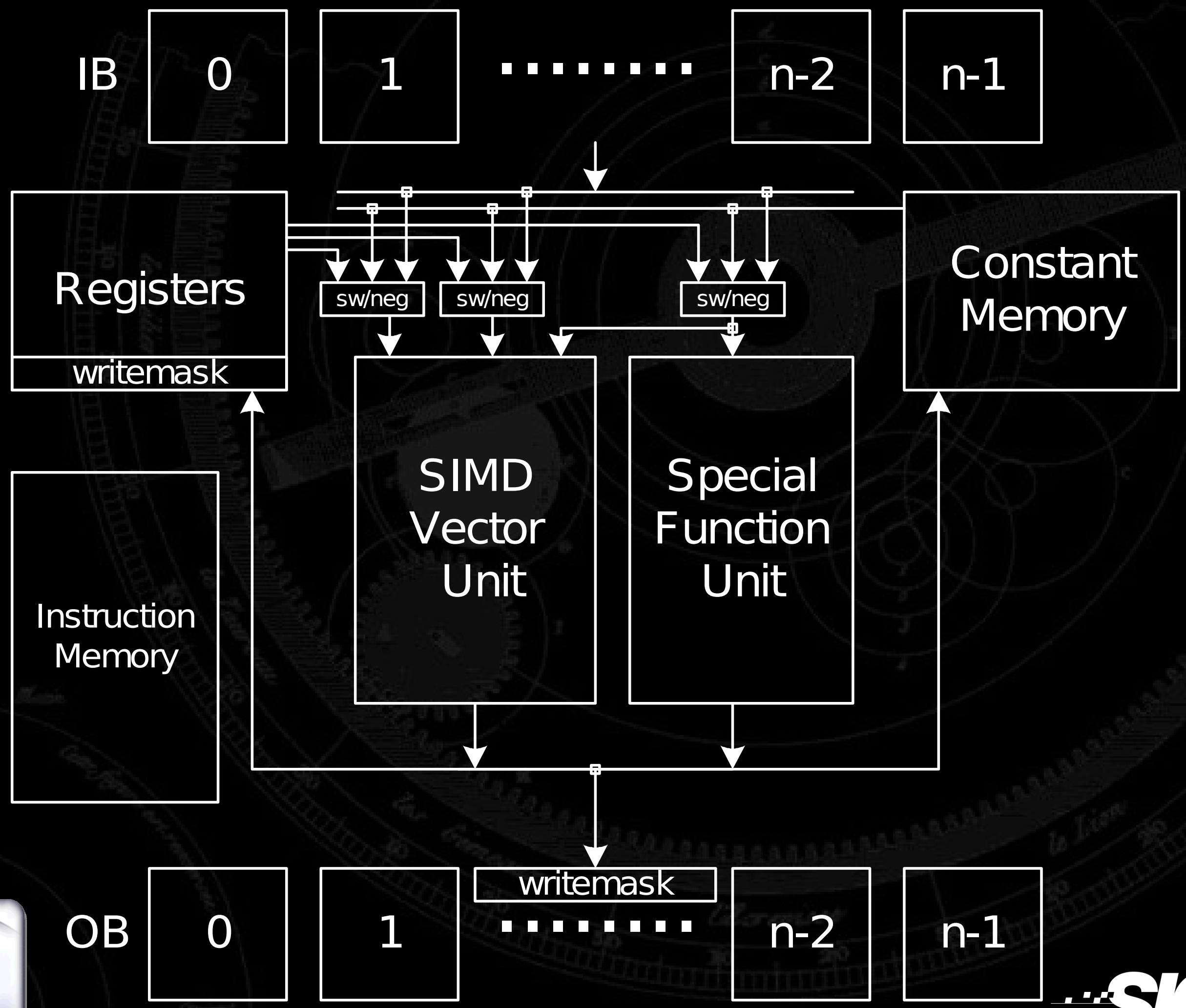
Instruction Set: The ops

- 17 instructions total
- ARL
- MOV, MUL, ADD, MAD, DST
- DP3, DP4
- MIN, MAX, SLT, SGE
- RCP, RSQ, LOG, EXP, LIT

Hardware Implementation

- Vector SIMD Unit + Special Function Unit
- Multithreaded and pipelined to hide latency
- Any one instruction/cycle
- All instructions equal latency
- Free swizzling/negate/write mask support

HW Block Diagram



API Support

- Designed to fit into OpenGL and D3D API’s
- Program mode vs. Fixed function mode
- Load and bind program
- Simple to add to old D3D and OpenGL programs

DEMOS

2001 EXPLORE INTERACTION AND DIGITAL IMAGES

Conclusion

- Very simple, efficient implementation
- Allows vertex programming continuity
- Stanford Imagine Architecture
- A work in progress, lots more to come…
- We welcome your feedback